Algoritmia e Desempenho em Redes de Computadores
1.º semestre - 2017/2018

Static Analysis of Inter-AS Routing

Rui Figueiredo (78247), Jorge Sacadura (78537), Alexandre Candeias (78599)

I. Introdução

O trabalho descrito consiste na implementação de um conjunto de algoritmos que permitem:

- Verificar se uma rede tem ciclos de clientes;
- Determinar se a rede é comercialmente conectada.
- Dado um nó de destino saber todos os tipos de ligação que cada nó conectado a ele tem.

A rede dado é armazenada num grafo, este grafo foi implementado com uma representação em lista de adjacências. Existem 3 tipos de ligação: *Customer*, *Peer* e *Provider* sendo que estas ligações são unidirecionais e estabelecem uma hierarquia na rede.

II. Implementação

Para a implementação tomou-se a decisão de representar a rede como um grafo direcionado, em que cada aresta tem um tipo que pode ser de um dos tipos de ligação enumerados anteriormente e definidos na figura 2. As ligações são armazenadas como uma lista de adjacências.

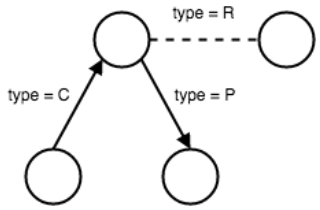

Fig. 1. Exemplo Tipos de ligação, que estão armazenados no grafo.

A. Ciclos de Cliente

O algoritmo para verificar se existem ciclos de cliente consiste em ver se o grafo tem ciclos quando apenas se considera as ligações correspondestes a *Provider-Customer* (Customer Links).

O algoritmo consiste em efetuar uma busca pelo grafo e marcar os nós com três estados, *visited*, *visiting* e *not visited* segundo algumas regras. A primeira vez que um nó é visitado é marcado como *visiting*, após todas as arestas que saem desse determinado nó terem sido analisadas o nó é marcado como visitado. O algoritmo é dado como concluído quando todos os nós estiverem marcados como *visited*, se o algoritmo alcançar este estado é porque o grafo não tem ciclos. Se a busca alcançar um nó que está em estado *visiting* significa que a busca já tinha passado por ali e encontramos um caminho para voltar aquele nó, como está em estado *visiting* é porque ainda há arestas que permitem sair daquele nó o que significa que encontramos um ciclo.

Apresenta-se então o pseudo-código para o algoritmo descrito, algoritmo 1 e 2. A complexidade do mesmo é a complexidade de uma DFS, logo $O(m+n)$ em que m é o número de nós e n é o número de ligações *Provider-Customer*.

Algorithm 1: hasCustomerCycles

```
input : Um grafo com a representação da rede
output: Um booleano a indicar se a rede tem ou não ciclos.
1  begin
2      foreach v do
3          states[v]:= NOT_VISITED
4      end
5      foreach v do
6          if states[v] is NOT_VISITED then
7              if visit(G, states, v) is TRUE then
8                  return TRUE
9      end
10 end
```

Algorithm 2: visit

```
input : O Grafo G, um array states com o estado de cada nó, e o
        nodeid, nó a visitar.
output: Um booleano True se encontrar cycles
1  begin
2      if states[nodeID] is VISITED then
3          return FALSE
4      if states[nodeID] is VISITING then
5          return TRUE
6      state[nodeID] := VISITING
7      foreach e in G.adj[nodeID] do
8          if e.type is CUSTOMER then
9              if visit(G, states, e.w) is TRUE then
10                 return TRUE
11     end
12 end
```

B. Rede Comercialmente Conectada

Uma rede é comercialmente conectada se, para cada destino, todos os nós elegem um ligação. Esta ligação pode ser de um de três tipos cliente, peer ou fornecedor.

O algoritmo implementado para verificar se uma rede é comercialmente conexa consiste em efetuar uma busca pelo grafo e no final verificar se todos os nós foram visitados, este é o algoritmo básico para verificar se um dado grafo é conexo. Como as arestas do grafo que representa a nossa rede tem tipos de ligação e existem regras para poder usar estas ligações só verificar que o grafo é conexo não é suficiente.

Durante a busca identificamos que todos os nós conseguem alcançar um nó (Tier-1), identificamos também quais são os nós (Tier-1) ou seja os nós que não têm fornecedores. Se esses nós possuírem uma ligação de *peer* entre eles isto é suficiente para garantir que a rede é comercialmente conexa, uma vez que todos os nós são direta ou indiretamente clientes de um Tier-1 e as rotas exportadas por clientes são exportadas para os *peer*.

O pseudo-código é apresentado nos algoritmos 3 e 4. A complexidade do algoritmo é $O(m+n+t^2)$ em que m é o número de nós, n é o número de arestas e t é o número de nós *Tier-1*, como o número de *Tier-1* normalmente é muito pequeno a complexidade resume-se a $O(m+n)$.

Algorithm 3: isComercialConnected

```
input : Um grafo com a representação da rede
output: Um booleano a indicar se a rede é o não comercialmente
        conexa.
1  begin
2      foreach v do
3          states[v]:= NOT_VISITED
4      end
5      S := {} ;                          ▷ Lista de nós Tier-1
6      search(G, states, k, S)
7      foreach v do
8          if states[v] is NOT_VISITED then
9              return FALSE
10     end
       ;     ▷ Identificar as conexões Peer para cada
         Tier-1
11     sets := []
12     foreach s in S do
13         foreach e in G.adj[s] do
14             if e.type is PEER then
15                 sets[s] := sets[s] ∪ e.w
16         end
17     end
       ;   ▷ Verificar se cada Tier-1 está ligado aos
         outros
18     foreach s in S do
19         foreach k in S do
20             if s not in sets[k] then
21                 return FALSE
22         end
23     end
24     return TRUE
25 end
```

Algorithm 4: search

```
input : Um grafo com a representação da rede G, o vetor com o
        estado dos nós,states, o nó a iniciar a visita,nodeID, lista
        dos tier-1 S
1  begin
2      states[nodeID]:= VISITED
3      isTier1:= TRUE
4      foreach e in G.adj[nodeID] do
5          if e.type is PROVIDER then
6              isTier1:= FALSE
7          if state[e.w] is NOT_VISITED then
8              search(G, states, e.w, S)
9      end
10     if isTier1 is TRUE then
11         S := S ∪ nodeID
12 end
```

C. Tipo de Ligações a um Nó de Destino

Foi implementado um algoritmo para saber quais os tipos de ligação que cada nó elege quando quer chegar a um nó de destino, o pseudo-código deste algoritmo está presente em 5.

O algoritmo implementado é uma versão de um *dijkstra* genérico onde as operações de seleção e composição são diferentes das usadas na versão habitual (*min* e $+$). O caminho dado pelo *dijkstra* é ótimo se estivermos na presença de um semi-anel $(S, \odot, \oplus, \Phi, \varepsilon)$.

No nosso caso como estamos num problema de *Routing* necessitamos apenas de ter um conjunto S , uma operação de seleção $\oplus$, um conjunto de mapas em S denotado por Γ, um elemento de identidade para $\oplus$ e a aniquilação para todos os mapas Γ que representamos por Φ.

O conjunto $S = \{c, r, d\}$, onde c representa rotas de *customer*, r de *peer* e p de *provider*. A operação de seleção($\oplus$) é dada na tabela I. O conjunto de mapas (Γ) é apresentado em II e a identidade da seleção e o aniquilador da composição Φ é dado por $\cdot$ que no nosso caso representa nenhuma rota eleita.

O operador $\oplus$ com a identidade $\cdot$ é seletivo pois respeita as seguintes propriedades:

- Seletividade:$a \oplus b = a$ ou $a \oplus b = b$ $\forall a, b \in S$, isto é facilmente visível pela tabela I. Nas seguintes propriedades quando nos referirmos a elementos a, b ou c é de notar que são $\forall a, b, c \in S$
- Associatividade:$(a \odot b) \odot c = a \odot (b \odot c)$. Para testar esta propriedade realizámos um *script* que para todas as possibilidades de elementos em S testa esta propriedade, o script está disponível nos ficheiros de código em anexo.
- Comutatividade:$a \odot b = b \odot a$. Esta propriedade é também visível na tabela I uma vez que a matriz que define o resultado do operador é simétrica.
- Identidade: $a \oplus \Phi = a$ com $\Phi = \cdot$, como é visível na 5ª linha e 5ª coluna da tabela.

É também importante garantir que o operador $\oplus$ garante ordem total em S, ou seja, $a \succeq b$ se $a \oplus b = a$. Para isso tem de garantir as seguintes propriedades.

- Reflexividade: $a \succeq a \equiv a \oplus a = a$. Isto é visível pela diagonal da matriz na tabela I.
- Transitividade: $a \oplus b = a$ e $b \oplus c = b$ então temos $a \oplus c = a$. Para testar esta propriedade recorremos mais uma vês a busca para todos os elementos de S.
- Anti-Simetria: $a \oplus b = a$ e $b \oplus a = b$ tem de implicar que $a = b$. A demonstração foi feita tal como anteriormente.
- *Least Element*: $a \oplus \Phi = a$. Isto é facilmente visível na tabela pois qualquer elemento operado com $\cdot$ o resultado é esse mesmo elemento.

A noção de ordem no nosso conjunto S irá então ser $c \succeq r \succeq p \succeq \cdot$, isto vai de acordo com a preferência de rotas que temos que é primeiro rotas de *customer*, em seguida *peer* e depois *provider*, por fim $\cdot$ que representa nenhuma rota disponível.

Além disto é importante definir que mapas Γ iram ser usados, estes mapas deve retratar o problema que queremos resolver, aqui é importante notar que o que queremos saber é como todos os nós chegam a um determinado destino. Estes mapas devem ainda satisfazer algumas propriedades:

- Aniquilação: $T(\Phi) = \Phi, \forall T \in \Gamma$. Isto é facilmente visível na tabela II
- Isotonicidade: $a \succeq b => T(a) \succeq T(b)$. Isto é facilmente verificável uma vez que em qualquer dos mapas á medida que vamos andando na coluna(mudando valor do domínio) da tabela II ficamos sempre com um elemento pior que o do argumento do mapa.
- Inflação: $a \succeq T(a)$. Esta propriedade é visível se para um dado valor do domino(colunas da tabela II) andando ao longo das linhas da tabela II, vemos que o que obtemos é sempre pior ou igual que o valor fixado na coluna.
- Next-Hop: $T(a) \succeq \Phi$ e $T(b) \succeq \Phi => T(a) = T(b)$. Isto é facilmente visível na tabela II, pois em todos os mapas apenas temos um valor diferente de $\cdot$.

$\oplus$	**c**	**r**	**p**	$\cdot$
c	c	c	c	c
r	c	r	r	r
p	c	r	p	p
$\cdot$	c	r	p	$\cdot$

TABLE I
OPERADOR SELEÇÃO

	c	**r**	**p**	$\cdot$
C	c	$\cdot$	$\cdot$	$\cdot$
R	r	$\cdot$	$\cdot$	$\cdot$
P	p	p	p	$\cdot$

TABLE II
MAPAS NO CONJUNTO S

Uma vez que são garantidas as propriedades acima discutidas, o algoritmo proposto converge para um caminho, e esse caminho é optimo.

Algorithm 5: electedRoutes

```
input : Um grafo com a representação da rede, o nó de destino
        (destination)
output: tipos de rota de cada nó para o nó de destino
1  begin
2      Q := ;                ▷ HEAP with not analised Nodes
3      foreach v do
4          d[v]:= NOROUTE
5          if v is destination then
6              d[v] := CUSTOMER
7          Q := Q ∪ v
8      end
9      while Q ≠ ∅ do
10         Q := Q - u ; ▷ Select a node u from Heap with
             highest priority
11         if d[u] is PROVIDER and Network is Commercially
             Connected then
12             Set the nodes left in Q to PROVIDER
13             break
14         foreach e in G.adj[u] do
15             if d[e.w] > exportedRoute(e.type,d[u])
                 then
16                 d[e.w] := exportedRoute(e.type,d[u])
17         end
18     end
19 end
```

Garantidas as propriedades essenciais convém explicar o que cada mapa faz aos elementos do conjunto S. Para isso recordamos que no algoritmo 5 quando um nó é retirado da fila ele é o que tem mais prioridade. Esta prioridade é definida de acordo com a operação de seleção discutida anteriormente. A rota que esse nó elege para o nó de destino é a guardada no vetor **d**,no momento em que é retirado do *Heap* e esse será o tipo de rota final. O que cada um dos mapas faz é mapear como é que cada um dos nós vizinhos consegue chegar ao nó destino passando pelo nó atual.

Este mapeamento depende de dois fatores. O primeiro é como é que o nó atual chega ao nó de destino, o segundo é qual o tipo de aresta que liga o nó atual ao seu vizinho. O tipo de aresta diz-nos qual o mapa que irá ser usado e a rota do nó atual diz-nos o argumento do mapa. Por exemplo caso a rota de um dado nó seja de *Customer* e o tipo de ligação da aresta a ser relaxada é *Provider* isto corresponde à linha **P** e coluna **c** da tabela II.

A dependência destes 2 fatores é retratada na tabela II, por exemplo, se o nó atual chegar ao destino como *provider* apenas são visíveis as ligações de *provider* e portanto apenas o mapa do tipo *provider* elege rotas (diferentes de $\cdot$). Ou seja, os nossos mapas retratam o protocolo de *routing*, as rotas que são ou não são exportadas. A operação de seleção elege a rota predileta quando existe mais do que uma ligação disponível, uma rota de *customer* é preferivel a uma de *peer* e uma de *peer* é preferível a uma de *provider*.

A função *exportedRoute* presente no algoritmo 5 indexa a tabela II e retorna o valor indexado. Na indexação *e.type* é o tipo de ligação (escolhe o mapa a utilizar) e *d[u]* é o tipo de rota do o nó atual para o nó de destino.

A complexidade do algoritmo proposto é no pior caso $O((n+m)log(n))$ onde n é o número de nós da rede e m o número de arestas. Esta complexidade é igual à do algoritmo Dijkstra pois o nosso procedimento é em todo semelhante. Para gerir a fila de prioridades implementou-se um *Heap*. A complexidade da operação de extrair o nó com prioridade mais alta bem como a de modificar a prioridade de um elemento é na nossa implementação $O(log(n))$.

Foi implementada um condição que permite terminar a procura sem ter de examinar todos os nós presentes no *Heap*. Como estamos a analisar os nós por ordem de acordo com a sua prioridade, tipo de rota que usam para chegar ao nó destino, se a rede for comercialmente conexa isto garante que primeiro iremos analisar os nós com rotas do tipo *Customer* em seguida *Peer* e por fim *Provider*. Deste modo podemos aquando da retirada do primeiro nó que elege uma rota de *Provider* para o destino podemos terminar o algoritmo, uma vez que de acordo com as operações descritas, todos os nós que se ligarem a nós com rota *Provider* vão ter rota *Provider*. Podemos então atribuir o tipo de rota *Provider* aos restantes nós que estão no fila de nós não analisados.

III. Resultados

A. Cycle

Desenvolveu-se um teste que consistia numa rede com ciclos de modo a testar se o nosso algoritmo identificava o ciclo ou não. O algoritmo conseguiu identificar o ciclo.

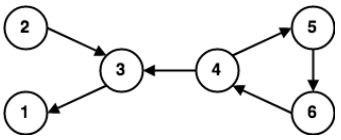

Fig. 2. Exemplo Tipos de ligação, que estão armazenados no grafo.

B. Rede Enunciado do Projeto

Para testar se o tipo de rota escolhida por cada nó está correta, executou-se o algoritmo para a rede apresentada no enunciado do laboratório. Os resultados são apresentados na tabela III, nas linhas apresenta-se o nó destino e nas colunas o nó origem. Os resultados vão de acordo com o esperado e deduzido teoricamente. Segundo o programa implementado a rede é comercialmente conexa e não existe *customer cycles*, o que vai de acordo com o esperado. Para esta rede efetuou-se ainda um teste que foi remover a ligação de *peer* entre os nós 10 e 11 e verificou-se como esperado que a rede deixou de ser comercialmente conexa.

D/O	1	2	3	4	5	6	7	8	9	10	11
1	c	p	p	p	c	p	p	p	p	c	r
2	p	c	p	p	c	c	r	c	c	c	c
3	p	p	c	p	p	c	c	c	c	c	c
4	p	p	p	c	p	r	c	p	c	r	c
5	p	p	p	p	c	p	p	p	p	c	r
6	p	p	p	p	p	c	r	c	c	c	c
7	p	p	p	p	p	r	c	p	c	r	c
8	p	p	p	p	p	p	p	c	p	c	r
9	p	p	p	p	p	p	p	p	c	r	c
10	p	p	p	p	p	p	p	p	p	c	r
11	p	p	p	p	p	p	p	p	p	r	c

TABLE III

TABELA DA ROTA ELEITA PARA CADA DESTINO POR CADA ORIGEM.

C. Ficheiro LargeNetwork

Obtemos os resultados da tabela IV em aproximadamente 8 minutos num Intel Core i7. É de notar que a ligação de um nó para ele próprio é contada como ligação de *customer*. Os resultados vão de acordo com o esperado, grande parte das ligações são ligações de *provider*. Este grande numero de ligações de provider faz com que ao terminarmos o algoritmo antes de analisar todos os nós tenha um grande ganho em termos de tempo de computação.

Tipo de Rota	%
Customer	0.5
Peer	20.5
Provider	79.0

TABLE IV

TIPOS DE LIGAÇÃO DE TODOS OS NÓS

IV. Conclusão

Através deste trabalho conseguimos ver como é possível através de um algoritmo que calcula caminhos mais curtos em grafos, como é o caso do Dijkstra, calcular caminhos em termos de *routing* dadas preferências entre rotas.

Através da definição de operações de seleção e composição que satisfazem as propriedades estudadas neste documento é possível também, adaptar o algoritmo proposto a outros tipos de problemas, como por exemplo calculo de caminhos ótimos em termos de capacidade ou outras propriedades do problema em estudo.

Por fim vimos que impondo algumas heurísticas especificas do problema, como o caso da terminação em caso da rede ser comercialmente conexa, tem grandes impactos no tempo de execução do algoritmo, portanto, conseguimos usar as propriedades intrínsecas do problema a nosso favor.